DESIGNING A REST API: LIBRARY CATALOGUE

https://github.com/ilya-ievlev/REST_design

the application must have entities book, author, category
simplified code of entities is given below

```
@Entity
@Table(name = "authors")
public class Author {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   private String name;

   @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, orphanRemoval =
true)
   private List<Book> books;
```

==========

```
@Entity
@Table(name = "categories")
public class Category {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   private String name;

   @ManyToMany(mappedBy = "categories")
   private Set<Book> books; // set was used because it shortens hibernate
query, making it faster in manyToMany relation
```

==========

```
@Entity
@Table(name = "books")
public class Book {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   private String title;

   @ManyToOne(fetch = FetchType.LAZY)
```

```
    @JoinColumn(name = "author_id")
    private Author author;

    @ManyToMany
    @JoinTable(
            name = "book_categories",
            joinColumns = @JoinColumn(name = "book_id"),
            inverseJoinColumns = @JoinColumn(name = "category_id")
    )
    private Set<Category> categories;
```

==========

Authorization

Authorization is handled via a JWT-based filter (JwtAuthFilter). The filter intercepts all HTTP requests and:

1. Extracts the token from the Authorization header (Bearer <token> format).
2. Validates the token.
3. If valid, allows the request to proceed.
4. If not, responds with 401 Unauthorized.

book api description

GET /books

- Returns paginated list of books
- Headers: Cache-Control: max-age=60
- Status: 200 OK
- Caching: Used — Read-only data that changes infrequently; improves response time and reduces load

GET /books/{id}

- Returns book by ID
- Headers: Cache-Control: max-age=60
- Status: 200 OK, 404 Not Found
- Caching: Used — Book data is read-heavy and does not change often; safe to cache for short duration

POST /books

- Creates a new book
- Headers: Location
- Status: 201 Created, 400 Bad Request
- Caching: Not used — Write operation must immediately reflect new data; caching could cause inconsistency

PUT /books/{id}

- Updates an existing book
- Status: 200 OK, 400, 404

- Caching: Not used — Updated content must be seen immediately by all clients; caching not allowed

DELETE /books/{id}
- Deletes a book
- Status: 204 No Content (success), 404 Not Found
- Caching: Not used — Deletion changes system state; caching could result in stale or incorrect responses

HATEOAS Usage
Each BookHateoasDto includes:
- self: /books/{id}
- author: /authors/{id}
- delete: /books/{id}

author api description

GET /authors
- Returns a paginated list of authors
- Query: page, size
- Headers: Cache-Control: max-age=60
- Status: 200 OK
- Caching: Used — Author list is relatively stable and read-heavy; caching improves efficiency

GET /authors/{id}
- Returns author details
- Headers: Cache-Control: max-age=60
- Status: 200 OK, 404 Not Found
- Caching: Used — Author data does not change frequently; suitable for short-term caching

POST /authors
- Creates a new author
- Headers: Location
- Status: 201 Created, 400 Bad Request
- Caching: Not used — Creation changes data state; must always hit the server to reflect updates

PUT /authors/{id}
- Updates existing author info
- Status: 200 OK, 400, 404
- Caching: Not used — Clients must receive up-to-date changes; caching is unsafe

DELETE /authors/{id}
- Deletes an author

- Status: 204 No Content, 404 Not Found
- Caching: Not used — Deletion alters system state; cached responses would misrepresent state

GET /authors/{id}/books
- Returns paginated list of books written by an author
- Query: page, size
- Headers: Cache-Control: max-age=60
- Status: 200 OK, 404 Not Found
- Caching: Used — Books by author are read-oriented; list can be cached safely for short durations

HATEOAS Usage
AuthorHateoasDto includes:
- self: /authors/{id}
- books: /authors/{id}/books
- delete: /authors/{id}

category api description

GET /categories
- Returns paginated list of categories
- Query params: page, size
- Headers: Cache-Control: max-age=60
- Status: 200 OK
- Caching: Used — Categories change rarely; safe to cache for short duration

GET /categories/{id}
- Returns details of a specific category
- Headers: Cache-Control: max-age=60
- Status: 200 OK, 404 Not Found
- Caching: Used — Category data is stable and suitable for caching

POST /categories
- Creates a new category
- Headers: Location
- Status: 201 Created, 400 Bad Request
- Caching: Not used — Creating new resources must reflect immediately; caching is inappropriate

PUT /categories/{id}
- Updates an existing category
- Status: 200 OK, 400, 404
- Caching: Not used — Data mutations must be reflected in real-time; caching disabled

DELETE /categories/{id}
- Deletes a category

- Status: 204 No Content, 404 Not Found
- Caching: Not used — Deletes must not be cached to ensure state consistency

GET /categories/{id}/books
- Returns paginated list of books in a category
- Query: page, size
- Headers: Cache-Control: max-age=60
- Status: 200 OK, 404 Not Found
- Caching: Used — Book listings under a category are not frequently modified

4. HATEOAS Usage
CategoryHateoasDto includes:
- self: /categories/{id}
- books: /categories/{id}/books
- delete: /categories/{id}

functional requirements:

User Authentication and Security
- All API endpoints must be protected using JWT authentication.
- Only authenticated users can access resources.

Entity Management (CRUD)
- Users can create, read, update, and delete:
  Books
  Authors
  Categories

HATEOAS Support
- All response DTOs must include HATEOAS links to related actions and resources.

Pagination
- GET endpoints returning lists (books, authors, categories, related books) must support pagination via page and size query parameters.

Content Format
- All requests and responses use application/json.
- Consistent DTOs must be used for all data transfer.

HTTP Status Handling
- Correct HTTP response codes must be used for all operations (e.g. 200, 201, 204, 400, 401, 404).

Caching
- Read-only endpoints (GET) must use short-term caching via Cache-Control headers.
- Mutation operations (POST, PUT, DELETE) must not be cached.

Error Responses

- Error responses must be consistent and descriptive.

nonfunctional requirements

Performance

- The API should handle high concurrency with minimal response time.
- Read operations should be optimized through caching and pagination.

Scalability

- The application architecture should allow scaling.

Security

- All endpoints must be protected via JWT.
- Tokens must be validated securely with expiration handling.
- Sensitive operations must be restricted to authenticated users.

Reliability

- The system must handle unexpected input and server errors gracefully.
- Proper HTTP status codes must be returned for all edge cases.

Maintainability

- Code should follow modular and clear structure (e.g., DTOs, services, filters).
- HATEOAS design must make the system self-explanatory for clients.

Extensibility

- The API design should support easy addition of new entities and endpoints.
- HATEOAS links help evolve the client-side logic without tight coupling.

Usability

- API responses must be consistent and predictable.
- Error messages should be descriptive and include cause where possible.

Portability

- The application must run on any servlet-compatible container (e.g., Tomcat).
- No dependency on platform-specific frameworks.

Documentation

- REST API behavior, supported status codes, and headers must be clearly documented.
- HATEOAS links must be included and described for each resource type.